Inteligencia artificial

Notas sobre la práctica 2: agentes delibertativos

E.T.S. de Ingenierías Informática y de Telecomunicación
Departamento de Ciencias de la Computación
e Inteligencia Artificial
Universidad de Granada

Curso 2023-2024

Notas sobre la práctica 2: agentes delibertativos

Índice

1. Objetivos

El objetivo de este documento consiste en hacer algunas aclaraciones sobre la realización de la práctica 2 de la asignatura. Las anotaciones tienen que ver tanto con la organización del código como con aspectos de funcionamiento de los agentes en los diferentes niveles propuestos.

2. Niveles

Los diferentes niveles a resolver en la práctica son los siguientes:

- 0: se implementa algoritmo de búsqueda en anchura. El agente que hay que guiar a la posición de destino es el jugador. Este nivel está resuelto mediante el tutorial de la práctica.

- 1: el algoritmo es el mismo que en el nivel anterior, pero ahora el agente a guiar hasta el objetivo es el colaborador.
- 2: debe usarse el algoritmo de coste uniforme. Esto implica que hay que tener en cuenta el coste de las acciones. Recordad que el coste de las acciones depende de la posición de partida del agente. En este nivel solo hay que preocuparse de guiar al jugador.
- nivel 3: vuelve a ser necesario guiar al colaborador, con la asistencia del jugador. En este nivel se pide implementar el algoritmo A^*. Para ello hemos de considerar que hay que agregar al coste una componente heurística. El valor de esta función heurística para un nodo concreto representa el coste de alcanzar la solución del estado de solución representado por el nodo. a partir del

3. Movimiento de los agentes

Conviene tener en cuenta las siguientes consideraciones:

- el colaborador se inicia con la orden CLB_STOP, de forma que está parado.
- para que el jugador sea capaz de dar órdenes al colaborador es preciso que lo tenga en su campo de visión. En este caso, puede ordenarle que realice cualquiera de las acciones disponibles para el colaborador. A partir de este instante, el colaborador puede seguir moviéndose, como un sonámbulo, aunque deje de estar en el campo de visión del jugador.

Conviene disponer de una función auxiliar que permita determinar si el colabodor está en el campo de visión del jugador. Hay que tener en cuenta que esta función debe tener en cuenta la orientación del jugador.

4. Consideraciones de diseño

4.1. Modularización del código

Las acciones necesarias para implementar los algoritmos de planificación de los diferentes niveles se repiten con frecuencia entre niveles. Para evitar tener código repetido conviene modularizar el programa y definir funciones que realicen tareas espcíficas. De esta forma, bastará con llamar a dichas funciones desde los niveles que se precisen.

4.2. Eficiencia

En los algoritmos de búsqueda se usan estructuras de datos para gestionar los nodos por explorar (frontera) y los nodos explorados. La gestión de estas estructuras debe ser eficiente para minimizar el tiempo de cálculo. En el tutorial, en la última versión, se trata este tema. Se proporciona una forma de comportamiento que trata de limitar el número de búsquedas en la lista de nodos por explorar (frontera).

Creo que es importante entender bien la forma en que funciona el código propuesto a este fin (similar al presentado en el tutorial):
La idea consiste en:

- limitar el número de comprobaciones al insertar nodos en la frontera.

Algoritmo 1 Forma de extracción de nodos de la frontera

```
// recupera el nodo en la primera posicion de la frontera
Nodo nuevo = frontera.front()
// borra el nodo en primera posicion para que no vuelva a considerarse
frontera.pop_front()
// si el nodo extraido de la frontera ya esta en explorados, se
// descarta y se extrae otro nuevo
while (!frontera.empty() and explorados.find(nuevo) != explorados.end()) do
    nuevo = frontera.front()
    frontera.pop_front()
end while
```

- al momento de extraer un nuevo nodo de la frontera para proceder a su análisis y generar nuevos nodos es cuando se comprueba si ya estaba incluido en la lista de explorados. En caso de estarlo, se descarta y se extrae otro nuevo.

4.3. Estructuras de datos para frontera y explorados

Las estructuras de datos a usar son:

- explorados: usaremos un objeto de tipo *set* en todos los niveles. Esta estructura de datos organiza los datos en un árbol binario, no permite repetidos. Para que funciones de forma correcta debemos implementar el operador $<$ en la estructura Nodo, ya que esos son los elementos que se almacenarán.
- frontera:
  - en los niveles 0 y 1 puede usarse la clase *list*.
  - en los niveles 2 y 3 debe usar algún tipo de datos que permita almacenar datos según un valor, en este caso el coste. Puede ser una cola con prioridad, por ejemplo.

Una posibilidad consiste en definir en la clase atributos para ambos tipos de estructuras de datos para la frontera (*fronteraLista* y *fronteraCola*, por ejemplo). Un método llamado *insertarFrontera* controla, según el nivel del juego, qué atributo usar en cada caso. Igual puede hacerse para el caso de *extraerFrontera*.

4.4. Organización de las operaciones básicas

Si dividimos el problema en tareas elementales, podemos distinguir las siguientes, que luego nos permitirán definir acciones más complejas, de más alto nivel, en base a ellas. Se presentan siguiendo este orden: de más básicas a más complejas.

Comenzamos considerando una tarea resposable de incrementar el plan mediante la incorporación de una nueva orden del jugador.

4.4.1. programar acción jugador

Este método recibirá como argumentos la acción a incluir en el plan y el nodo actual que está en exploración. Se indican únicamente los argumentos que resultan imprescindibles, pero podéis usar otros adicionales si os resulta necesario. El pseudocódigo se indica a continuación

Algoritmo 2 programar acción jugador(accion, actual, ...)

```
// el método devuelve un nuevo nodo obtenido de aplicar la acción sobre el nodo actual

// creación del nuevo nodo como copia del actual
Nodo nuevo = actual

// se aplica la accion sobre el nuevo nodo (este método ya está implementado en el tutorial)
nuevo.estado = aplicar(accion, nuevo.estado, mapa)

// si el nuevo estado es válido (es decir, pudo realizarse la acción)
if nuevo estado es válido then

    // se comprueba la última orden del colaborador. Si no era CLB_STOP se aplica
    if la última orden del colaborador no es CLB_STOP then

        // se aplica la última acción del colaborador sobre el estado producido al aplicar
        // la orden del jugador
        nuevo.estado = aplicar(acción, nuevo.estado, mapa)

        // si la acción es válida (pudo aplicarse y el nuevo estado es diferente de actual)
        // se calcula el coste y se inserta en la frontera
        si es necesario, calcular el coste
        incluir acción en la secuencia de acciones del plan
        insertar nuevo en frontera
    end if
end if
```

4.4.2. programar acciones del jugador

Este método hará uso del anterior. Su responsabilidad consistirá en considerar todas las posibles acciones a realizar por el jugador y para cada uno de ellas llamará al método anterior. El psuedocódigo puede describirse de la siguiente forma (se observa que se indica que debe recibir como argumento el nodo en análisis):

Algoritmo 3 programar acciones jugador(actual, ...)

```
// programación de la orden actRUN
nuevo = programar acción jugador (actRUN, actual, ...)

// se comprueba la consecución del objetivo para no seguir analizando
comprobar objetivo conseguido(nuevo, ...)

// programar actWALK
if objetivo NO conseguido then
    // programacion de la orden actWALK
    nuevo = programar acción jugador(actWALK, actual, ...)

    //comprobar si el objetivo se ha conseguido
    comprobar objetivo conseguido(nuevo, ...)
end if

// programar actTURN_L
if objetivo NO conseguido then
    // programar acciones de giro: por si mismas nunca producen la consecución
    // del objetivo, pero podría conseguirse por las acciones del colaborador
    nuevo = programar acción jugador(actTURN_L, actual, ...)

    // comprobar si el objetivo se ha conseguido
    comprobar objetivo conseguido(nuevo, ...)
end if

// programar actTURN_SR
if objetivo NO conseguido then
    // no hace falta comprobar objetivo conseguido al no haber más acciones que programar
    nuevo = programar acción jugador(actTURN_SR, actual, ...)
end if
```

4.4.3. programar acción colaborador

Esta es la función equivalente a la de programación acción para el jugador. El pseudo código sería así:

Algoritmo 4 programar acción colaborador(accion, actual, ...)

```
// el método devuelve un nuevo nodo obtenido de aplicar la acción sobre el nodo actual

// creación del nuevo nodo como copia del actual
Nodo nuevo = actual

// si la acción coincide con la última acción realizada entonces
// se programa la acción actIDLE para el jugador
if acción == última acción del colaborador then

    // programar acción actIDLE para el jugador
    programar acción jugador(actIDLE, nuevo, ...)
else
    // se aplica la acción pasada como argumento
    nuevo.estado = aplicar(acción, nuevo.estado, mapa)

    // se actualiza la última orden del colaborador
    nuevo.estado.ultimaOrdenColaborador = acción

    // si la acción pudo realizarse, actualizar plan y se inserta en la frontera
    if la acción es válida y se produce cambio en el estado then
        incluir acción en secuencia de acciones del plan
        insertar nuevo en frontera
    end if
end if
```

4.4.4. programar acciones colaborador

Usando como herramienta auxiliar el método anterior, podemos definir ahora el comportamiento del método encargado de explorar las posibles acciones del colaborador:

Algoritmo 5 programar acciones colaborador(actual, ...)

```
// programación de la orden act_CLB_WALK
nuevo = programar acción colaborador (act_CLB_WALK, actual, ...)

// se comprueba la consecución del objetivo para no seguir analizando
comprobar objetivo conseguido(nuevo, ...)

// programar act_CLB_TURN_SR y act_CLB_STOP
if objetivo NO conseguido then
    // programacion de la orden act_CLB_TURN_SR
    nuevo = programar acción colaborador(act_CLB_TURN_SR, actual, ...)

    // programacion de la orden act_CLB_STOP
    nuevo = programar acción colaborador(act_CLB_STOP, actual, ...)
end if
```

4.4.5. búsqueda para nivel 1

Usando las funciones anteriores podemos describir ahora la estructura general del método de búsqueda en anchura para el nivel 1:

Algoritmo 6 búsqueda en anchura para nivel 1

```
insertar nodo inicial en frontera

// bucle de búsqueda
while frontera no vacía y objetivo no conseguido (jugador) do
    // extracción de nodo de frontera
    actual = extraer nodo frontera

    // insertar actual en explorados
    insertar actual en explorados

    // comprobar si actual representa una solución
    if actual NO es solución then
        // programar acciones del jugador si el colaborador no está en su campo de visión
        if colaborado NO en campo de visión del jugador then
            programar acciones del jugador
        else
            programar acciones del colaborador
        end if
    end if
end while
```

4.4.6. búsqueda para nivel 2

El código es muy parecido al anterior, pero ahora solo tendremos que considerar las acciones del jugador y la frontera debe ser una estructura tipo cola, de forma que al extraer un nuevo nodo para explorar sea aquel de menor coste:

Algoritmo 7 búsqueda en anchura para nivel 2

```
insertar nodo inicial en frontera

// bucle de búsqueda
while frontera no vacía y objetivo no conseguido (jugador) do
    // extracción de nodo de frontera
    actual = extraer nodo frontera

    // insertar actual en explorados
    insertar actual en explorados

    // comprobar si actual representa una solución
    if actual NO es solución then
        // programar acciones del jugado
        programar acciones del jugador
    end if
end while
```

4.4.7. búsqueda para nivel 3

El código es exactamente igual al del nivel 1, considerando que debe usarse para la frontera una cola con prioridad, igual que ocurre en el nivel 2.

Otra diferencia en este nivel consiste en la necesidad de ampliar el coste con el valor de una función heurística. Mi recomendación consiste en usar un único tipo de estructura Nodo, a la que se le agregan todos los campos que deseemos (y se usarán o no dependiendo del nivel de juego).

Como se avanzó antes, la heurística aporta una estimación del coste necesario para alcanzar la solución del problema a partir del nodo considerado. Como propuesta inicial puede usarse la diferencia en valor absoluto entre las filas y columnas del jugador y colaborador con respecto a la posición objetivo (en valor absoluto).